

Introspective and Adaptive Runtime Systems with Transactions

Rodric Rabbah
IBM Research

Saman Amarasinghe
MIT CSAIL

Transactions afford boundaries during the lifetime of a program run where it may be convenient to *introspect* – inspect the state of execution and the immediate history leading to it – and *adapt* subsequent computation.

As an example, imagine if each transaction carried a utility attribute that can be evaluated when the transaction is issued or is ready to commit. If the transaction results in a conflict and its utility is considered sufficiently low, then rather than reconciling the conflict, it may be more worthwhile to simply ignore it altogether. In this example, it is assumed that each transaction refines the goal of the algorithm, and needless refinements can be safely ignored. Studies [1, 2] have shown that many applications, from desktop to scientific codes, are often naturally resilient to some errors during computation.

More formally, the underlying thesis is the following: *the programming model should allow for malleable transaction granularities, and should not impose a rigid set of policies on how transactions are issued, executed, and committed. Rather, the programming model should allow programmers that understand how their program execution should be orchestrated to conveniently relay their insights to the runtime system and architecture, and without disturbing the algorithm expression.* In other words, the argument is for a two-level programming model consisting of a *bridge* and an *engine room*. The bridge provides the programming language that is used for describing the user algorithms in an architecture independent manner. The engine room describes a set of policies that orchestrate the execution of the algorithms on the target architectures.

```
while (beliefs have not converged) {
  foreach n in nodes {
    foreach e in edges(n) {
      compute message to neighbor(e, n)
    }
  } transaction<xnode> with
    utility { !converged(n) }

  foreach n in nodes {
    belief = prior distribution
    foreach e in edges(n)
      belief *= message from
        neighbor(e, n)
  }
}
```

```
Iterator<xnode> xiter;
while (!xiter.end())
{
  // get any pending transactions
  xaction = xiter.any();

  // evaluate utility function
  if (xaction.utility() < useful)
    abort xaction;
  else
    do xaction;

  xiter.remove(xaction);
}
```

Example bridge (left) and engine room (right) programs for a loopy belief propagation algorithm.

The engine room may also adjust the granularity of the transactions (e.g., replace the **xnode** iterator with an iterator over **edges**), and the conditions under which to abandon one transaction granularity in favor of another.

In essence, the bridge and engine room allow for a separation of concerns. The engine room is a pluggable scheduler that allows users to orchestrate the execution of their transactions to the extent they care to constrain, relax, or manipulate it. The engine room does not muddle the clean algorithm details described in the bridge. The runtime system can provide a set of default policies that the engine room overrides. For architectures with a rich set of heterogeneous ingredients, the scheduler also provides a natural boundary for mediating the mapping of transactions to resources, under direct user control, or even through dynamic optimizations.

The multilevel programming approach can extend beyond transactions, and is a more broadly applicable framework for introspection and adaptation. It borrows ideas from sketching [3] and telescoping languages [4]. It can also extend beyond performance tuning, to aid in program debugging, runtime verification, and security.

References:

- [1] [Enhancing Server Availability and Security Through Failure-Oblivious Computing](#), Rinard, Cadar, Dumitran, Roy, Leu, and Beebe. *OSDI*, December 2004
- [2] [Application-Level Correctness and its Impact on Fault Tolerance](#), Li and Yeung. *ISCA*, February 2007.
- [3] [Programming by Sketching for Bit-Streaming Programs](#), Solar-Lezama, Rabbah, Bodik, and Ebcioğlu. *PLDI*, June 2005.
- [4] [Telescoping Languages: A System for Automatic Generation of Domain Languages](#), Kennedy, Broom, Chauhan, Fowler, Garvin, Koelbel, McCosh, and Mellor-Crummey. *IEEE*, Volume 93, Issue 3, February 2005.

Acknowledgements: DARPA program on Architectures for Cognitive Information Processing, and the CEARC team.